Designing an Accessible Web Technology

by

Jacob Mouka

Submitted to OCAD University

in partial fulfillment of the requirements for the degree of

Master of Design

in

Inclusive Design

Toronto, Ontario, Canada, August 2013

Author's Declaration

Acknowledgements

I would like to thank Colin Clark (Inclusive Design Research Centre, OCAD University), Richard Schwerdtfeger (IBM), Jutta Treviranus (Director, Inclusive Design Research Centre, OCAD University), and Peter Coppin (OCAD University) for their time and invaluable guidance.

Abstract

This project considered the limitations of accessibility in web technology and screen readers. It was an attempt to create a framework for building web pages and applications that would have accessibility built in and make development easier. It also involved building a prototype screen navigator that demonstrated ways of overcoming the shortcomings of current screen readers. The final demonstration was an email web client built using this framework. The purpose of the email web application was to explore the viability, benefits and limitations of the framework's method of creating web applications, and to test the usefulness of the prototype screen navigator. The findings for the framework were that it has benefits, for both the users of assistive technologies and developers, but there remain gaps, ideas and questions for further exploration. The prototype navigator made interacting with the dynamic application fairly easy and efficient.

Table of Contents

List of Tables

List of Figures

Introduction

Most of the work in web accessibility focuses on bridging the gaps, but the current strategy of adapting web pages to assistive technologies (AT) has limitations: it is reactive, it is an extra (and often left out) step in the development process, and the results tend to be a compromise. This project explores whether it is possible to design a web technology that is inherently accessible, makes it easier for developers to create complex web pages, and has the ability to customize the user interface for different users' needs.

The project attempts to achieve built-in accessibility by creating a semantic, simple data layer that AT can interact with, as the base for the architecture. The architecture is also a framework for adding graphical user interfaces (UI) to this semantic layer. The exposed semantic layer would be an interface that all kinds of devices and AT could interact with to optimize the experience for their users. In other words, the semantic layer would be a base for building graphical applications, as well as an intermediate interface for AT. In this context, AT is anything that customizes the experience for the user beyond the developer's original intent (e.g., providing alternate interactions for eye-tracking systems, or emphasizing primary content by changing colour or contrast). Using such an architecture could make building accessible applications transparent to developers because they would not have to do extra work to achieve accessibility.

The main principle of this project is to build web applications in a way that explicitly separates content (and its interactions) from the UI and code that drives it. This is based on the software engineering pattern Model-View-Controller (MVC). The goal of this architecture is to divide web pages into an essential base layer and an extended layer (e.g., graphical UI). The essential layer of an application is its data, its core functionality and the minimum HTML needed to present and interact with that data. It should be thought of as the base application that can be extended with various UIs. The goals are to give developers more flexibility and make it easier to build complex web pages, and to allow AT to interact directly with a simpler but complete essential layer, and have the flexibility to modify the UI.

One technology that is similar to this MVC framework is XSL ("The Extensible Stylesheet Language Family (XSL)," n.d.). XSL includes a language (XSLT) to transform XML documents into other documents such as HTML or plain text. Its goals are similar to this project in that it separates content from presentation and provides flexibility for creating multiple representations. There are several differences between these projects. The aim of this project is to make it simple for non-developers to use, while XSLT is a fairly complex programming language. Another difference is that XSLT generates new documents from XML documents, while this MVC framework is intended to be used for modifying dynamic web pages in real time.

This project has three major parts. The first part was creating a framework for building web pages that allows for this style of MVC architecture. The second part was looking at major limitations of current screen readers, and building a prototype screen navigator that demonstrates how assistive technology can benefit from web pages created using the framework. The screen navigator demonstrates effective ways for non-visual interaction with a dynamic web page. The final part was building an email web application using this framework to test the usefulness of the prototype screen navigator and explore the ease of development and the benefits and limitations of building applications using this method.

The work of this project is an exploration of different possible architectures for the web platform. The strategy was to imagine an ideal situation and work backwards to try to implement it. Some of the findings show value, while others demonstrate gaps and point to new ideas and future work.

The MVC Framework

The concept for the framework was based in part on Cascading Style Sheets (CSS), which allow developers to create the style of a web page separately from the page itself. This increases flexibility and improves accessibility by limiting how much unnecessary information the assistive technology needs to process. But CSS is limited when it comes to visually laying out a web page. Achieving a specific visual layout requires the document to be structured in a certain way, which excludes other visual layouts. It also cannot extend the functionality of the web page. This framework attempts to build a similar system to CSS, but one that allows templates to be loaded to reorganize the web page and allows developers to inject functionality to drive the UI. An important goal is to place as few restrictions on the base page as possible (ideally none), so that any page can be developed independently of this framework.

This framework uses a similar approach to that of the Mercator system created by (Mynatt, 1992). Their approach was to build a method of monitoring graphical applications built for the X Windows System (used to build graphical applications for various Linux and Unix systems). The system's widgets have an analogous off-screen model that focuses on structure and semantics. The mapping of the widget hierarchy to the off-screen counterpart is similar, but not one-to-one. The widget hierarchy includes many elements that are too low-level to be useful to the off-screen model. The goal was to simplify the structure and provide more semantics for the screen reader. Users could navigate the graphical UI by following the semantic structure of the off-screen model. This allows them to build a mental model of the application and navigate more intuitively and efficiently.

The Mercator system translated an application's graphical components (e.g., windows, buttons and text boxes) into the semantic off-screen model by monitoring the running application (via the client-server communication and Editres protocol of X Windows) to determine its graphical hierarchy and the user's interactions (Mynatt & Edwards, 1992).

This formed a general solution for translating any application built using the X Windows toolkit to a screen reader. Thus, implementing screen reader accessibility was transparent to the application's developers.

One advantage that this MVC framework has over the Mercator system is full access to the DOM (document object model). X Windows limited the amount of information (about both the application's structure and the user's interaction) available to Mercator. The Mercator system also could not access any elements built using methods other than the X Windows toolkit (e.g., custom drawing routines). The framework's unrestricted access to a web page's DOM expands its potential.

The MVC framework in this project has three main components. The first includes the syntax and rendering engine for transforming the web page. The second is a system for reacting to dynamic updates of the web page. The final component is a system extending the functionality of the web page. The framework was built using JavaScript and the jQuery library.

The rendering system uses a combination of plain HTML and custom tags for templates that define the structure of widgets and page elements (see *Figure 1* for a sample). These templates are similar to CSS rules but define the structure of HTML nodes instead of style. The templates are loaded from external files, and in this way the visual layout of a page can be changed at any time, exactly like CSS; however, unlike CSS, the system also has the ability to wrap, re-arrange and modify HTML nodes of the base page, as well as insert new nodes. Therefore, the web page can be re-organized considerably. The syntax was designed to be easy to learn for anyone familiar with HTML. The framework was built with existing application frameworks and development practices in mind. For example, extra care was taken not to break references to HTML nodes that might be managed by other frameworks. In this way, this framework can be used to extend capabilities of existing application frameworks.

```
<layout for="#widget">
   <div id="widgetWrapper">
      <node selector=".name"></node>
      <div class="widgetBody">
         <node selector="img" onclick="clickItem(this)"></node>
         <node selector=".delete"
      onbefore="confirmDelete"
      onafter="animateDelete"></node>
         <node selector=".view" onafterajax="animateMainArea"></node>
      </div>
   </div>
   <edit selector="a" class="widgetLink"></edit>
</layout>
```

***Figure 1.* Sample template used by the framework to define the structure of widgets and page elements. The syntax allows developers to re-arrange, insert and update page elements, and inject additional functionality.**

The framework uses a system of observing the DOM for dynamic updates and triggering actions as needed. This is similar to the way CSS rules are applied to dynamically updated nodes. The framework uses the DOM4 Mutation Observer interface (Anne van Kesteren, Aryeh Gregor, Lachlan Hunt, & Ms2ger, 2012) to receive notifications when nodes are created or deleted. When a node that matches a template is created, it needs to be updated by the rendering engine. When a node with a template is deleted, additional cleanup is performed. The developers of the base document do not need to do anything extra to allow templates to be loaded later, and the templates are automatically applied in response to dynamic updates.

The final component of the framework is a system for extending the functionality of the base page. The goal is to inject code that drives and animates the UI added by the templates. The base page is responsible for content and its code, and the templates are responsible for the UI and its code. The framework is the piece that controls the two sides, in a true MVC architecture.

It has two main methods of injecting additional functionality. The first uses the jQuery event system ("Event Handler Attachment | jQuery API Documentation," n.d.). The template syntax allows for triggering code before and after the code already associated with an element. Additional UI updates can be performed when a user interacts with some page element. It is also possible to inject code before and after an asynchronous (AJAX) action.

The second method is a registry that allows the template code to receive notifications for specific DOM updates. It can be difficult to determine what specific nodes were updated by an action in the base code, and this registry enables the template's code to react to those updates. For example, if the base code has some background task (e.g., AJAX) that updates the DOM, the template's code can register to be notified of those updates (e.g., when a node is created or deleted), and react accordingly. Please refer to *Figure 2* for sample code.

```
mvcController.registerInsertAndDeleteCallback('folder',
      initFolderList, removeFolder);
mvcController.registerOnShow('[role=main]', richAppLogin);
```

Figure 2. **Sample code of the framework's registry. It monitors the DOM for updates matching the CSS selector (first parameter) and executes the passed-in functions. In this sample, when a *folder* node is created or deleted, it triggers *initFolderList* and *removeFolder,* respectively. Similarly, it will execute *richAppLogin* when any *[role="main"]* item is shown.**

Prototype Screen Navigator

This project looked at three major gaps in current screen readers. The first is that it is difficult for users to create a mental model of the web page. The main strategy people use is to look at headings, and to a lesser extent labels, skip links and descriptive anchors (Bigham, Cavender, Brudvik, Wobbrock, & Lander, 2007). These elements have been found to significantly improve navigation, but they only give an approximation of the web page's structure, and only when used properly. The second gap is that it is difficult to skim and skip content. Skipping content meaningfully is related to the difficulty of building an accurate mental model of the page, as well as the facility to navigate the page. It has been found that headings give rough landmarks, but generally users of screen readers use a scanning technique, jumping ahead and back, trying to zero in on content (Takagi, Saito, Fukuda, & Asakawa, 2007). Finally, it is difficult for users to interact with dynamic web pages (Bigham et al., 2007)(Brown, Jay, Chen, & Harper, 2012). The W3C

Web Accessibility Initiative - Accessible Rich Internet Applications (WAI-ARIA) specification significantly improves screen reader interaction with dynamic content, but many developers do not use it properly or do not use it at all ("WebAIM: Screen Reader User Survey #4," 2012). It also requires special attention to be correctly implemented (Scheuhammer, Cooper, Pappas, & Schwerdtfeger, 2013), and it can be difficult to engineer effective user interfaces for AT (Brown & Harper, 2013).

The prototype screen navigator built in this project demonstrates how an AT can interact directly with a web page's DOM to overcome these gaps. It uses a similar strategy to that of the Mercator system (Mynatt, 1992); it interacts with the structure and semantics of the content, rather than the graphical UI. While the Mercator system had limited access to the DOM (due to limitations of X Windows), this prototype has full access to the DOM. The framework allows the screen navigator to ignore loading of the UI layer and interact with a very simple but functionally complete version of the web page.

To enable the user to create an accurate mental model of the web page, the screen navigator navigates by following the structure of the content (DOM hierarchy). As the user navigates the page (see Table 1 for the navigation scheme), they build an accurate mental model of the content's structure. This hierarchical navigation makes it possible to skip content meaningfully and skim the page effectively and efficiently (Treviranus, n.d.). It also makes it fairly easy for a user to keep track of where they are on the page. Future work could focus on different ways of giving the user more context, such as reading out the full path from the root node to the current selection.

The screen navigator attempts to meaningfully describe the current selection using a variety of methods. It first describes the node itself. If the node has a WAI-ARIA role, the screen navigator uses that; otherwise, it states the kind of tag in plain language (e.g., "link" for <*A*>). If the node contains a sub-hierarchy, the screen navigator mentions the number of child nodes. Finally, it peeks into the hierarchy to find the first element with visible content and reads it out. There are other ways to describe the node, and future versions could offer customization.

Navigation Keys	**Function**
up and down arrow keys	move the selection to the previous and next items, respectively, in the current sub-hierarchy (i.e., sibling nodes)
right arrow key	drill into the sub-hierarchy of the current selection
left arrow key	navigate back up the hierarchy
?	describe the current selection
page up and page down keys	attempt to find the previous and next similar elements, respectively, in a list of repeated items (e.g., the "to" field in a list of emails)
enter key	activate the current selection (equivalent to the left mouse button), or interact with a text input element

***Table 1*. Navigation scheme for the prototype screen navigator.**

As an aside, this project used XML to describe the structure of the page's data. For example, for emails it used an <*EMAIL*> tag, with <*SUBJECT*>, <*FROM*>, etc., tags for the email's data. This way, the screen navigator can accurately describe the data structure (e.g., “from: john@website.com”). There are several reasons for and against using XML, and varying opinions on how it could be used in web pages, and this is a discussion that should be continued. For example, should developers be allowed to create arbitrary XML structures for their data? How can we maintain semantics and interoperability, and also ensure flexibility and ease of use? For the purpose of this project, XML was a simple, effective way of defining data structures and enabling the screen reader to accurately describe them.

The screen reader uses the MVC framework’s DOM observer facility to respond to dynamic page updates. When page elements are created or made visible, the screen reader is notified by the framework and announces that a node has been changed (e.g., “Notification for item just shown: email”). The user can open a notifications menu and browse the list of notifications. They are marked (and announced) as *new* until the user has selected them, and they are ordered newest to oldest. Selecting a notification makes the user’s selection jump to the updated node, where they resume navigation. They can jump back to the selection previous to the selected notification item. Thus, the screen reader has a second mode of navigation based on the history of selection jumps. For example, in the email web application, the user can click the “reply” link on an email, which causes the email composer form to appear. They can select the form from the notification menu, fill it out, press the “send” link, and use the jump history to return to the email they just replied to. Page elements that are removed or hidden do not require notification unless they are the user’s current selection. In those cases, the screen reader announces that their selection has been removed (or hidden) and selects the nearest parent node, to allow the user to keep their page orientation.

This system is an effective way to interact with a dynamic web page. Furthermore, it does not require any special work by the developer. By interacting directly with the DOM, the screen reader has an accurate account of the events on the page. Future work could focus on notification style, customization of alerts (e.g., ignoring certain elements), etc.

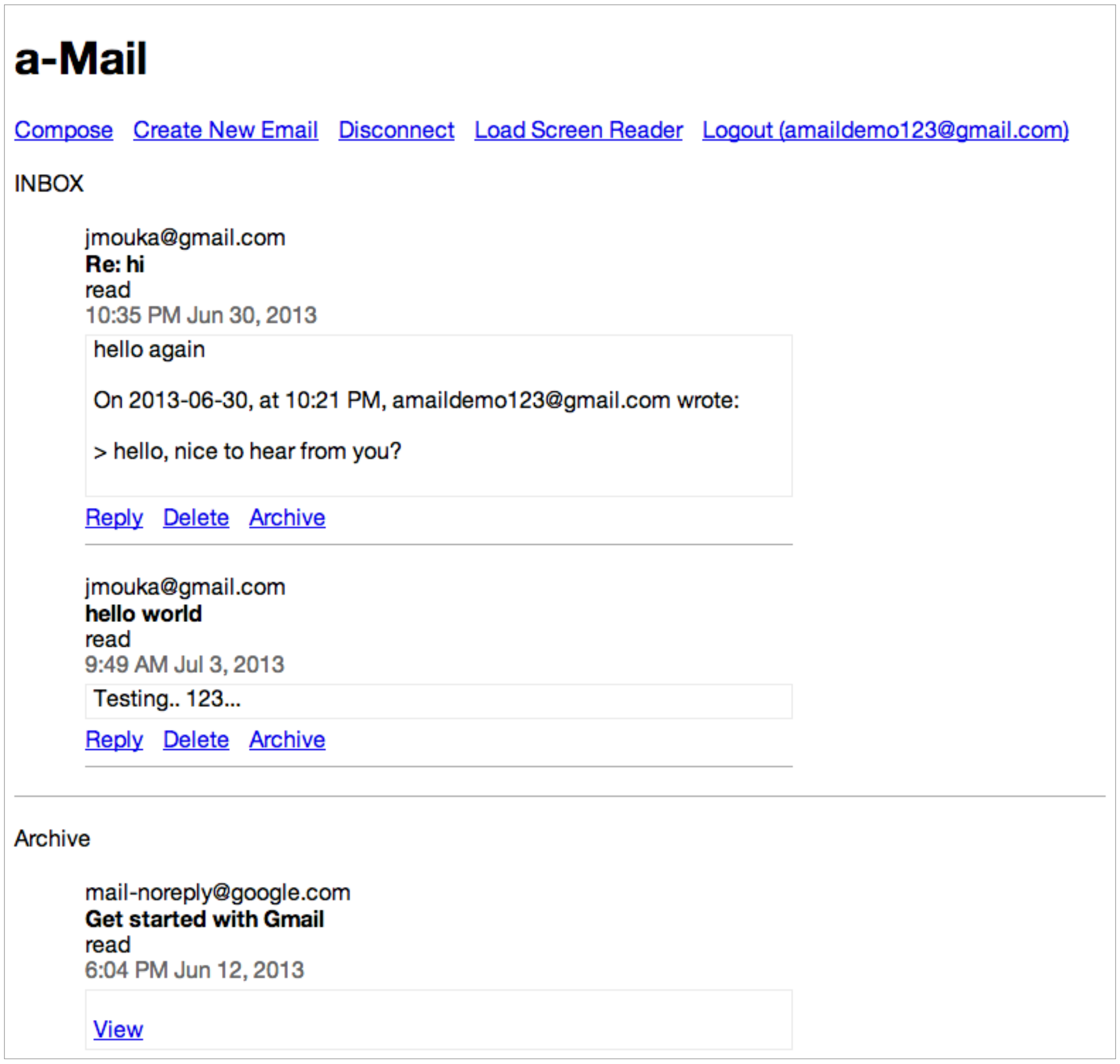
The screen reader has a text entry mode for interacting with input elements. When such an element is selected, pressing *enter* changes modes from navigation to text entry, and the keyboard reverts to the default behaviour. Pressing *escape* returns the user to navigation mode. The intention was to separate the activities (navigation and text entry) to avoid interference.

Email Web Application

The third part of this project was creating a working web application using this framework. The purpose was to test the usefulness of the screen reader and explore the viability, ease of development, and benefits and limitations of developing applications using the framework. An email web client was chosen because it is a familiar application, it has enough dynamic interactivity to be a fairly representative interactive web page, and a basic working example was relatively simple to implement.

The plain page (*Figure 3*) contained the application's essential functionality and had only basic styling using CSS. This is the essential base layer, focusing on content and its interactions. The screen reader was then loaded and used to test interacting with the application (*Figure 4*). Finally, the framework was used to create the interactive graphical UI for the application by loading templates and code (*Figure 5*). The templates changed the application's UI considerably and used many of the framework's features. Some examples are:

- the page structure is re-arranged considerably, to include a header, sidebar, main area, etc.
- email navigation is re-organized into folders, with buttons for selecting folders, the sidebar for listing emails in the folders, and the main area for displaying the selected email
- extra functionality is injected to animate certain actions; for example, deleting an email animates the action in the sidebar (the email shrinks, then disappears) and hides the main area
- the template's code uses the framework's DOM observer to update the folder buttons when the base code creates or deletes folders


a-Mail

Compose Create New Email Disconnect Load Screen Reader Logout (amaildemo123@gmail.com)

INBOX

jmouka@gmail.com
Re: hi
read
10:35 PM Jun 30, 2013

hello again

On 2013-06-30, at 10:21 PM, amaildemo123@gmail.com wrote:

> hello, nice to hear from you?

Reply Delete Archive

jmouka@gmail.com
hello world
read
9:49 AM Jul 3, 2013

Testing.. 123...

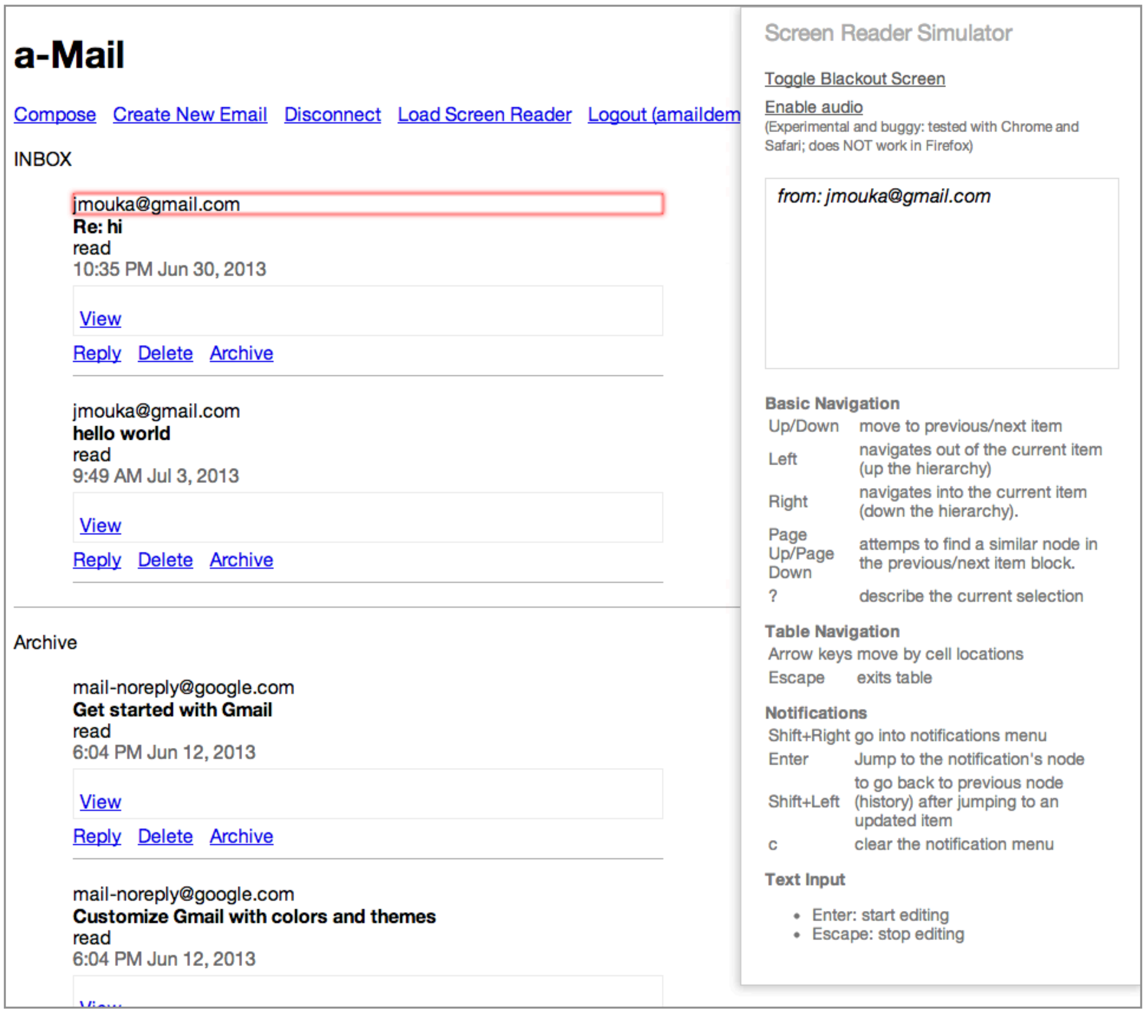
Reply Delete Archive

Archive

mail-noreply@google.com
Get started with Gmail
read
6:04 PM Jun 12, 2013

View

***Figure 3.* Screenshot of the plain (base) page of the email client (a-Mail, for Accessible Mail).**



a-Mail

Compose Create New Email Disconnect Load Screen Reader Logout (amaildem

INBOX

jmouka@gmail.com
Re: hi
read
10:35 PM Jun 30, 2013

View

Reply Delete Archive

jmouka@gmail.com
hello world
read
9:49 AM Jul 3, 2013

View

Reply Delete Archive

Archive

mail-noreply@google.com
Get started with Gmail
read
6:04 PM Jun 12, 2013

View

Reply Delete Archive

mail-noreply@google.com
Customize Gmail with colors and themes
read
6:04 PM Jun 12, 2013

Screen Reader Simulator

Toggle Blackout Screen

Enable audio
(Experimental and buggy: tested with Chrome and Safari; does NOT work in Firefox)

from: jmouka@gmail.com

Basic Navigation

Up/Down	move to previous/next item
Left	navigates out of the current item (up the hierarchy)
Right	navigates into the current item (down the hierarchy).
Page Up/Page Down	attemps to find a similar node in the previous/next item block.
?	describe the current selection

Table Navigation

Arrow keys move by cell locations	
Escape	exits table

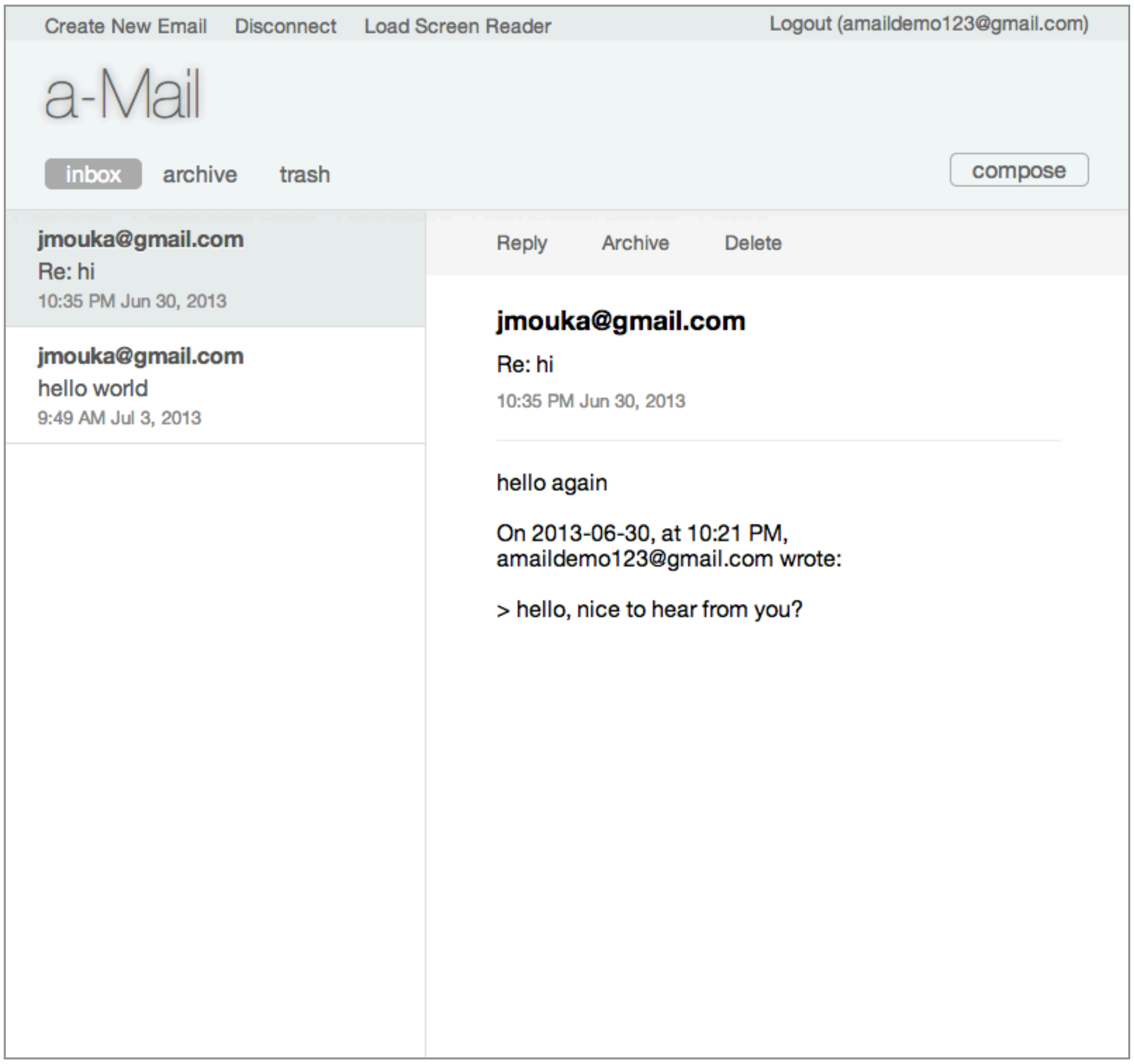
Notifications

Shift+Right	go into notifications menu
Enter	Jump to the notification's node
Shift+Left	to go back to previous node (history) after jumping to an updated item
c	clear the notification menu

Text Input

- Enter: start editing
- Escape: stop editing

Figure 4. **Screenshot of the plain page with the screen navigator. The *from* field is selected and described in the screen navigator's panel (right).**



jmouka@gmail.com
Re: hi
10:35 PM Jun 30, 2013

jmouka@gmail.com
hello world
9:49 AM Jul 3, 2013

jmouka@gmail.com
Re: hi
10:35 PM Jun 30, 2013

hello again

On 2013-06-30, at 10:21 PM,
amaildemo123@gmail.com wrote:

> hello, nice to hear from you?

Figure 5. **Screenshot of the email client with a graphical user interface.**

Results

The MVC framework worked well in loading a rich, interactive graphical UI onto the plain email client. The rendering engine was capable of creating the graphical UI, and the system of extending functionality (both injecting functionality onto page items, including

AJAX requests, and using the DOM observer) proved sufficient to create the rich, interactive application. From building the email client, it was found very valuable to split the development into phases focusing on the content and the presentation. The flexibility afforded by being able to add templates without having to change the base page was also valuable. The template syntax was easy to use and afforded a lot of flexibility in UI development. The framework has the potential to save developers a considerable amount of work. In the email client, 53% of the total HTML was for the graphical UI (content was 52 lines of HTML, and UI templates were 58 lines of HTML). Reusing such templates on a multi-page website would mean significant savings. Many of these results are qualitative and are the author's reflections. The method was to compare developing web pages using this framework to using plain HTML, CSS and JavaScript.

The results for the prototype screen navigator were very positive. Navigating the web page's logical DOM structure was found to be efficient, and it was easy to track one's location. It was particularly valuable knowing explicitly what the selected item was (e.g., the *subject* node in an email) and preventing the cursor from unintentionally navigating to other sections. It was also easy to skip content and find specific items. The notification system for dynamic updates worked well, and it was easy to react to alerts, e.g., to read a new email or reply to an email. Particularly, it was useful to be able to jump directly to the notification's associated node, interact with it and use the jump history to jump back. These results are the author's reflections and are qualitative comparisons to existing screen reader technology. For example, using Google's Gmail web application with

VoiceOver on Mac OS X 10.8, it was difficult to build a mental model of the web page and the structure of the page elements, even with the use of WAI-ARIA landmarks. It was also difficult to keep track of the current selection, and it was difficult to stay in the desired section. Dynamic updates were not obvious and their nature was not clear (e.g., updating the email list after choosing a folder). A lot of time was spent navigating over content to find the desired item (e.g., browsing the list of emails).

For example, browsing the email list of the email client using the screen navigator was compared to using VoiceOver with Google's Gmail. For Gmail, it was easy to use the WAI-ARIA landmark to find the emails section, but browsing the email list required serial navigation over all elements, many of them UI elements such as checkboxes and images whose function was not clear. It was not clear what email element (e.g., subject, date) was selected, since VoiceOver simply read the text on the screen. It was not clear where one email ended and another began, and the content's structure had to be extrapolated (e.g., each email starts with a checkbox item). There was no easy way to skim the email list. In comparison, *Table 2* shows an example of navigating the email list with the screen navigator. The key points were: the content's organization was explicit, which made it easy to move between sections; an email's elements (e.g., subject, date) were explicitly described; it was clear where an email started and ended; and it was easy to skim the list.

User’s Action	Example of Screen Navigator Output
Select main section (third item on page, i.e., press down arrow twice to find it, and right arrow to select)	“main section”
Select a folder, e.g., inbox (press down arrow to find the folder, and press right arrow to select)	“folder: 2 items. First: INBOX”
Select the folder’s emails (press down arrow to find emails section, and right arrow to select)	“emails, 5 items”
Select an email list (press up arrow or down arrow to browse the list, and right arrow to select an email)	“email, 8 items: First: jmouka@gmail.com”
Browse an email (press up arrow and down arrow to select the email’s elements)	“subject: Re: hi there!”
Return to the emails list (press left arrow)	“email, 8 items. First: jmouka@gmail.com”
Use up arrow and down arrow to select the previous or next email, respectively	

***Table 2*. Example of steps performed to browse the list of emails in the prototype screen navigator.**

Discussion

The goal of the framework was to place minimal restrictions on the base page and code. This was not completely possible. The main restriction on the base code is that it could not assume a specific hierarchy. For example, when an email’s *delete* link was pressed, that link could not assume its direct parent was the main email container. Instead, the nodes must perform a hierarchy search up the parent chain to find the desired nodes. This should not present a major hurdle, but it was a required programming pattern. The query also added a performance penalty, but in practice the hierarchies were found to be small (three or fewer nodes long), and there is no branching when moving up in the hierarchy, so this penalty is small. These queries were also fairly infrequent, since they are the results of a user's actions. On the other hand, there are no such limitations for application frameworks that maintain direct references to nodes, as is a common architecture. Similarly, the templates loaded cannot remove any nodes expected by the base code or re-arrange the structure in a way that would break the base code. This is a small limitation since the architecture is such that the templates are allowed to depend on the base page. In practice, this was not a limitation, as the logical structure of the base page generally followed a similar structure in the templates, typically only adding, re-ordering or hiding nodes. This is a matter of best practices.

Two development patterns and questions about structure were noted in developing the email web application. The first pattern is that it is difficult for the template code to directly determine which nodes were changed by the base code. Chaining functionality was ineffective in this since the base code does not return references to the DOM updates. The pattern was to register with the framework's DOM observer to be notified of specific updates (e.g., when a new email node is created) and trigger additional functionality.

The second pattern was a strategy of searching for appropriate points in the base page's workflow to inject additional functionality. For example, to delete an email in the graphical application, does it make sense to create a separate delete button in the main view area, which executes code specific to the rich version and then triggers the base page's delete link? Or is it better to inject functionality into the base page's delete link and have the user click it directly? In general, the template code does not interact directly with the base code, and instead the developer looks for the best places to inject additional functionality.

Finally, it was not always clear how best to structure the content to be logically organized, or even what elements should be part of the base page. The base page contains a login form and an email composer form, and it was not obvious where those forms should be on the page. For example, in an early version of the email client, the email composer form was dynamically inserted below the *reply* link when that was pressed. This made sense in the page's workflow, but the resulting page structure was not logically organized. This could be confusing to people using the screen navigator and it could be too limiting for developers to work with. Instead, it was better to make these forms siblings to the other main sections (navigation and main area) and show/hide them as needed.

The focus of this project was presenting an architecture for building rich, interactive web applications that allow assistive technology to interact with a simpler, essential layer of the web page. In this narrow view, it was successful. However, it was found difficult to generalize this method to other assistive technology, such as keyboard-only navigation of the graphical user interface. The problem seems to stem from the fact that HTML is ultimately a mixture of content and presentation elements. One strategy that was attempted was to mark the base page's elements as content and make use of this information, but this was undone whenever page elements were created dynamically and not updated by the framework. For example, in the email application, the main email viewer area was generated by the template code, so the framework would mark it as presentation, but a person using keyboard-only navigation would not be able to reach this area when following content-only nodes.

The prototype screen navigator showed that interacting with the content, rather than a complex graphical UI, was an effective way of interacting with a dynamic web page. The content's structure and semantics made building a mental model of the page fairly easy, which enabled skimming and efficient navigation. The prototype's method of observing the DOM allowed it to react to dynamic updates directly, without special considerations from the developer. Similar research is being done by others (Brown & Harper, 2013). The next step would be to perform user testing, to test these claims further.

The screen navigator was a simple prototype with many potential areas for improvement. One area is the ability to customize the notification alerts (what information to present, frequency of alerts, types of alerts, ability to ignore, etc). The current notification system could become unusable due to too many dynamic updates. The prototype was also limited in how it reacted to some dynamic updates. For example, when the selected node was deleted or hidden, the navigator selected the parent node and notified the user. This may not be optimal behaviour, and further testing is required.

Future Directions

It seems that it would be very valuable if content could always be distinguishable from presentation, regardless of how it was generated or structured. There are several strategies that could be explored in future work. It could be useful to have two distinct sets of tags for each category. This would solve some of the issues mentioned earlier, but could make development quite cumbersome. Another possible strategy is to build a system where the DOM is composed entirely of content nodes, and the visual layout is created in a separate layer. For example, CSS could be expanded to include sophisticated layout and graphical rules. This has the potential to achieve the goals of this framework, but it is not clear if such a system would be too complicated to use.

Future work on the screen navigator could include further optimizing web page interactions to meet different users' needs. For example, classifying dynamic updates and providing specialized ways of dealing with them, and customizing the way different devices interact with a web page by using the event model being developed by the W3C Indie UI group (Craig & Cooper, 2013).

Conclusion

This project involved developing a framework for building web pages and dynamic web applications that offered flexibility to developers and had accessibility built-in. The goal was to explore whether accessibility can be a fundamental part of the web platform, instead of requiring additional steps. It also explored major gaps in screen readers and included a prototype screen navigator that demonstrates how assistive technologies could interact with dynamic web pages built using this architecture. The goal of the screen navigator was to explore more efficient ways for screen readers to interact with dynamic web pages.

The approach for the framework was to separate content (and its interactions) from the presentation (and code to drive it). The framework made it easy to add user interfaces to the content layer. The content layer was left exposed to the user's device to allow optimization of the user experience. In this way, the content layer was a base for graphical applications as well as an interface for all kinds of devices and assistive technology. The main contribution of this project was to build a prototype of this architecture in HTML, CSS and JavaScript, and explore its benefits, development patterns, ease of use and gaps. The findings were that this architecture was effective at building dynamic, graphical web applications, and offered flexibility and efficiency to the developer. Two development patterns were identified in building web pages with this architecture. The project found several gaps in the prototype: the content layer had restrictions on how nodes were referenced, and there was a small performance penalty. It was also not obvious how best to organize the content layer. This architecture provided an effective interface for a screen navigator to interact with dynamic content; however, it was difficult to generalize this approach to other assistive technologies. Two approaches that have the potential to overcome these gaps were presented for future study.

The main finding of the prototype screen navigator was that it is possible to have efficient, non-visual interaction with a dynamic web page. Navigating a web page by following the hierarchical structure of the content allowed for building an accurate mental model of the content, efficient navigation, easier tracking of one's position, and providing explicit information about the content. It also included an effective notification system for dynamic updates of the web page, and presented future testing and enhancements of the system. The main contribution of this prototype was to demonstrate how assistive technology could optimize the experience for the user when the AT has direct access to the content layer of a web page. It also showed that accessibility could be achieved without requiring extra steps from the web page's developer.

This project demonstrated benefits and gaps in the presented architecture, and it presented several directions for future study.

References

Anne van Kesteren, Aryeh Gregor, Lachlan Hunt, & Ms2ger. (2012, December 6). DOM4 - W3C Working Draft. Retrieved July 9, 2013, from http://www.w3.org/TR/dom/

Bigham, J. P., Cavender, A. C., Brudvik, J. T., Wobbrock, J. O., & Lander, R. E. (2007). WebinSitu: a comparative analysis of blind and sighted browsing behavior. In *Proceedings of the 9th international ACM SIGACCESS conference on Computers and accessibility* (pp. 51–58). New York, NY, USA: ACM. doi:10.1145/1296843.1296854

Brown, A., & Harper, S. (2013). Dynamic injection of WAI-ARIA into web content (p. 1). ACM Press. doi:10.1145/2461121.2461141

Brown, A., Jay, C., Chen, A., & Harper, S. (2012). The uptake of Web 2.0 technologies, and its impact on visually disabled users. *Universal Access in the Information Society*, *11*(2), 185–199.

Craig, J., & Cooper, M. (2013, January 22). IndieUI: Events 1.0. Retrieved July 10, 2013, from http://www.w3.org/TR/indie-ui-events/

Event Handler Attachment | jQuery API Documentation. (n.d.). Retrieved July 6, 2013, from http://api.jquery.com/category/events/event-handler-attachment/

Mynatt, E. D. (1992). Auditory Presentation of Graphical User Interfaces. In *Proceedings of the 1992 International Conference on Auditory Display* (pp. 533–555). Addison-Wesley Publishing Company.

Mynatt, E. D., & Edwards, W. K. (1992). Mapping GUIs to auditory interfaces (pp. 61–70). ACM Press. doi:10.1145/142621.142629

Scheuhammer, J., Cooper, M., Pappas, L., & Schwerdtfeger, R. (2013, February 25). WAI-ARIA 1.0 Authoring Practices. Retrieved July 9, 2013, from http://www.w3.org/WAI/PF/aria-practices/

Takagi, H., Saito, S., Fukuda, K., & Asakawa, C. (2007). Analysis of navigability of Web applications for improving blind usability. *ACM Trans. Comput.-Hum. Interact.*, *14*(3). doi:10.1145/1279700.1279703

The Extensible Stylesheet Language Family (XSL). (n.d.). Retrieved January 16, 2013, from http://www.w3.org/Style/XSL/

Treviranus, J. (n.d.). Nimble Document Navigation Using Alternative Access Tools. Retrieved January 15, 2013, from http://www.ra.ethz.ch/cdstore/www6/Access/ACC202.html

WebAIM: Screen Reader User Survey #4. (2012, May). Retrieved July 9, 2013, from http://webaim.org/projects/screenreadersurvey4/